Von Hackern und Holländern

Microcontroller-Code-Optimierung mit Bittwiddling und Kombinatorik

Martin Hilscher

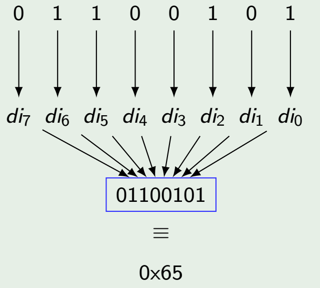
Gegeben

- 64 digitale In-Ports
- Jeder Port liefert ein Bit eines *long*-Werts

Gesucht

- Funktion *bitscan*(*long bitset*)
- Führe für jedes 1-Bit *process*(*i*) aus
- Mit *i* Index des gesetzte 1-Bits
- Reihenfolge ist unwichtig

Beispiel (8 Bit)



```java
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Wieviel kann man da schon falsch machen?

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
     if ((bitset >>> i) & 1 == 1) {
       process(i);
     }
   }
}
```

Fast alles !!!111!!

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Wird für jedes Bit aufgerufen

Fast alles !!!111!!

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Wird für jedes Bit aufgerufen

Im Schnitt nur für jedes 2te Bit

Fast alles !!!111!!

$64 \cdot 4$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Fast alles !!!111!!

$64 \cdot 4$ cycle

$64 \cdot 1$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Fast alles !!!111!!

$64 \cdot 4$ cycle

$64 \cdot 1$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle

Fast alles !!!111!!

$64 \cdot 4$ cycle $64 \cdot 1$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle $64 \cdot 1$ cycle

Fast alles !!!111!!

$64 \cdot 4$ cycle $64 \cdot 4$ cycle $64 \cdot 1$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle $64 \cdot 1$ cycle

	Plain
Ohne Loop-Unrolling:	704 cycle

Fast alles !!!111!!

$64 \cdot 4$ cycle

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle

$64 \cdot 1$ cycle

	Plain
Ohne Loop-Unrolling:	704 cycle
Mit Loop-Unrolling:	384 cycle

Fast alles !!!111!!

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

	Plain	Branch Prediction
Ohne Loop-Unrolling:	704 cycle	+ ?
Mit Loop-Unrolling:	384 cycle	+ ?

Fast alles !!!111!!

```
public static void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Für diesen Code
extrem kontraproduktiv

	Plain	Branch Prediction
Ohne Loop-Unrolling:	704 cycle	+ ?
Mit Loop-Unrolling:	384 cycle	+ ?

Zwei Ideen

1. Wenig (/ Keine) bedingte Sprünge
   - $\Rightarrow$ Vermeidet teure Operationen
   - $\Rightarrow$ Branch-Prediction ausgeschaltet
2. Weniger Rechnen
   - Berechne schnell Position des nächsten 1-Bits
   - Überspringe die 0-Bits
     - $\Rightarrow$ I can haz speedup !!!111!!

Finde das nächste 1-bit

Erster Schritt

0010 0001 0100 0100	x		
1101 1110 1011 1011	˜x		
1101 1110 1011 1100	˜x + 1	≡	-x
0000 0000 0000 0100	x & (˜x + 1)	≡	x & -x

Finde das nächste 1-bit

Erster Schritt

0010 0001 0100 0100	x		
1101 1110 1011 1011	~x		
1101 1110 1011 1100	~x + 1	$\equiv$	-x
0000 0000 0000 0100	x & (~x + 1)	$\equiv$	x & -x

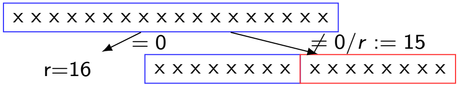
Problem

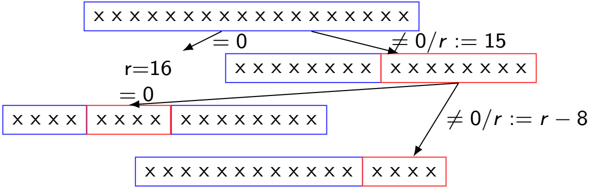
Jetzt haben wir 2^i, aber wir brauchen i.

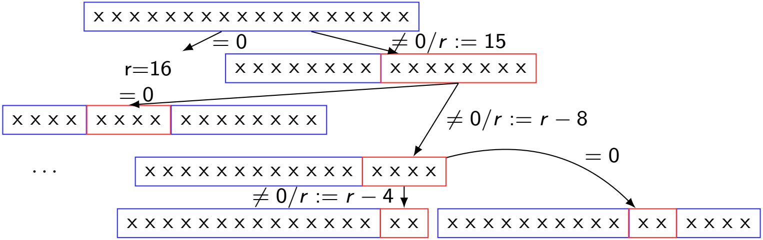
$2^i \longrightarrow i$ zähle rechtsstehende 0 bits

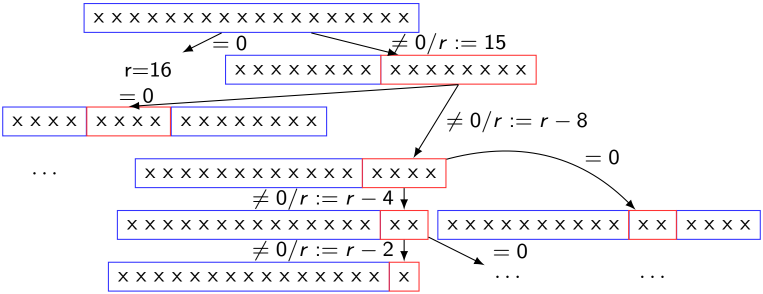
- Wir zählen einfach die 0-Bits rechts vom 1-Bit
- Können nicht alle Bits einzeln Zählen
  - $\Rightarrow$ Für jedes gesetzte Bit der Algorithmus von Folie 2
- Benutze binäre Suche

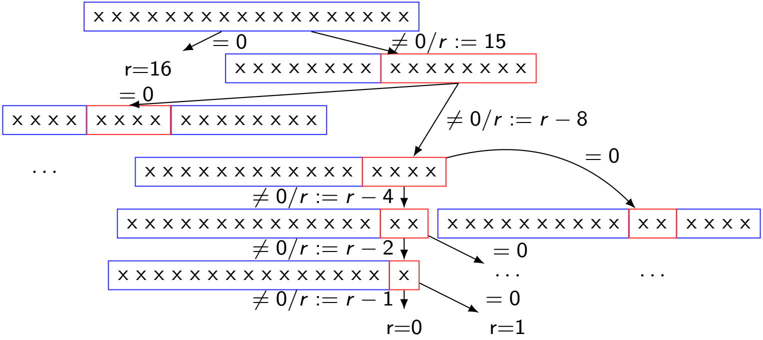
x x x x x x x x x x x x x x x x x x


x x x x x x x x x x x x x x x x x x
= 0
≠ 0/r := 15
r=16
x x x x x x x x
x x x x x x x x


x x x x x x x x x x x x x x x x x x
= 0
≠ 0/r := 15
r=16
x x x x x x x x x
x x x x x x x x x
= 0
x x x x
x x x x
x x x x x x x x x
≠ 0/r := r − 8
x x x x x x x x x x x x x
x x x x


x x x x x x x x x x x x x x x x x x
= 0
≠ 0/r := 15
r=16
x x x x x x x x x
x x x x x x x x x
= 0
x x x x
x x x x
x x x x x x x x x
≠ 0/r := r − 8
= 0
. . .
x x x x x x x x x x x x x
x x x x
≠ 0/r := r − 4
x x x x x x x x x x x x x x x
x x
x x x x x x x x x x
x x
x x x x


= 0
r=16
≠ 0/r := 15
= 0
≠ 0/r := r − 8
...
= 0
≠ 0/r := r − 4
≠ 0/r := r − 2
= 0
...
...


x x x x x x x x x x x x x x x x x x x
$= 0$
$\neq 0/r := 15$
r=16
x x x x x x x x x x x x x x x x x x
$= 0$
$\neq 0/r := r - 8$
x x x x x x x x x x x x x x x x x x x x x x x x x
. . .
x x x x x x x x x x x x x x x x x x
$= 0$
$\neq 0/r := r - 4$
x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x
$\neq 0/r := r - 2$
$= 0$
. . .
. . .
x x x x x x x x x x x x x x x x x x x
$\neq 0/r := r - 1$
$= 0$
r=0
r=1

```java
public static int numberOfTrailingZeros(long i) {
  int x, y;
  if (i == 0) return 64;
  int n = 63;
  y = (int)i;
    if (y != 0) { n = n - 32; x = y; }
    else x = (int)(i >>> 32);
  y = x << 16; if (y != 0) { n = n - 16; x = y; }
  y = x <<  8; if (y != 0) { n = n -  8; x = y; }
  y = x <<  4; if (y != 0) { n = n -  4; x = y; }
  y = x <<  2; if (y != 0) { n = n -  2; x = y; }
  return n - ((x << 1) >>> 31);
}

public static void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(Long.numberOfTrailingZeros(t));
    bitset ^= t;
  }
}
```

Probleme

- Immer noch zu viele bedingte Sprünge :-(
- 33 Operationen / 1-Bit :-C

Probleme

- Immer noch zu viele bedingte Sprünge :-(
- 33 Operationen / 1-Bit :-C

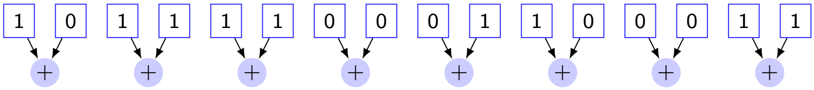
Aber ...

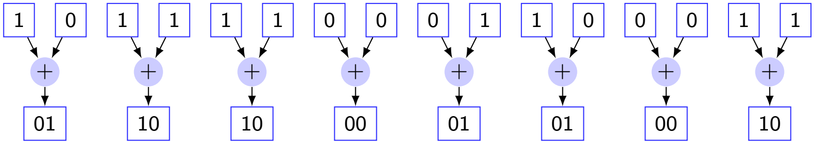
- Wir betrachtent nur einen Spezialfall
- Es gibt noch andere Lösungen

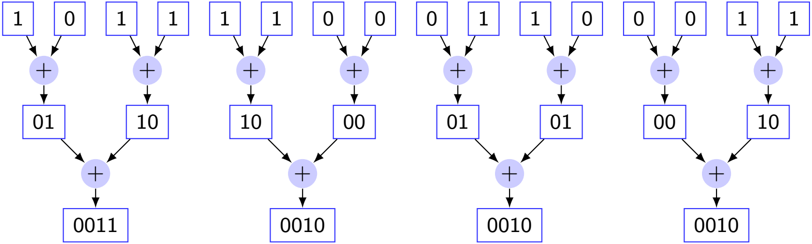
$2^i \longrightarrow i$ nutze $2^i - 1 = 2^{i-1} + 2^{i-2} + \ldots + 2^0$

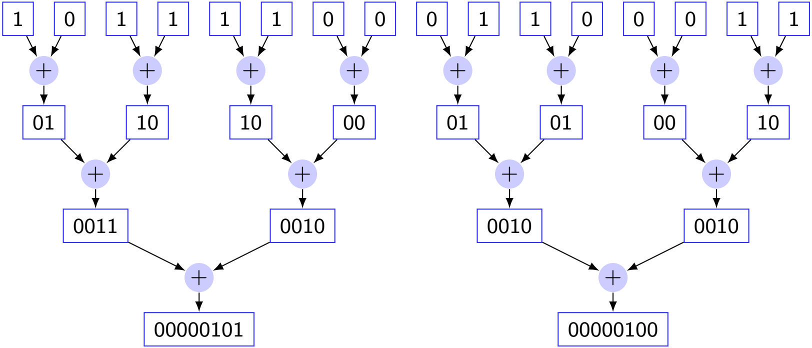
- Wir betrachten 2^i und suchen i
- Es gilt: $2^i - 1 = 2^{i-1} + 2^{i-2} + \ldots + 2^0$
- Oder binär z.B.: $2^8 - 1 = 11111111_b$
- Also: $\#_1(2^i - 1) = i$
- Wir müssen also nur die 1-Bits zählen

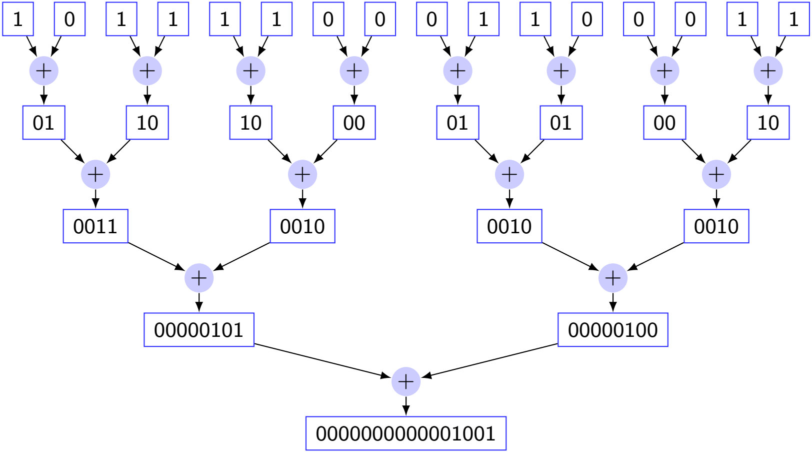
1	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1


1
0
1
1
1
1
0
0
0
1
1
0
0
0
1
1
+
+
+
+
+
+
+
+

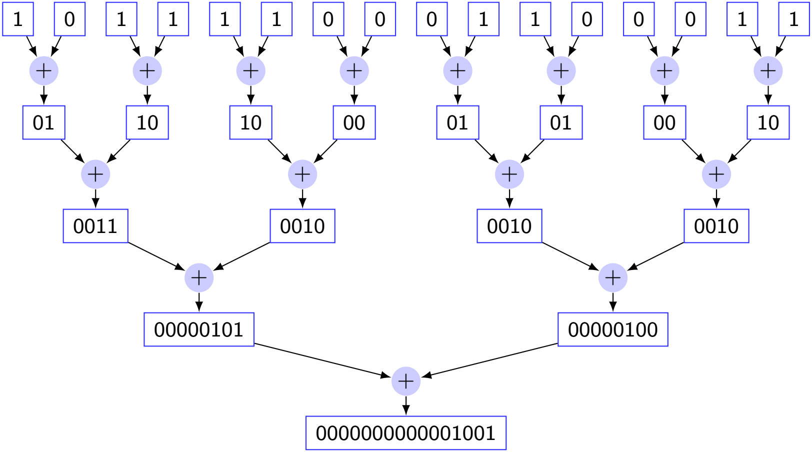

1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1
+ + + + + + + +
01 10 10 00 01 01 00 10


1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1
+ + + + + + + +
01 10 10 00 01 01 00 10
+ + + +
0011 0010 0010 0010


1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1
01 10 10 00 01 01 00 10
0011 0010 0010 0010
00000101 00000100


1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1
01 10 10 00 01 01 00 10
0011 0010 0010 0010
00000101 00000100
0000000000001001

Wieviele Operationen sind das?



Schnelle parallele Addition von Bits

- $$\begin{array}{ll} 10\ 11\ 11\ 00\ 01\ 10\ 00\ 11 & \mathrm{x} \\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01 & \mathrm{0x5555} \\ \hline 00\ 01\ 01\ 00\ 01\ 00\ 00\ 01 & \mathrm{x}\ \&\ \mathrm{0x5555} \end{array}$$
- $$\begin{array}{ll} 10\ 11\ 11\ 00\ 01\ 10\ 00\ 11 & \mathrm{x} \\ 01\ 01\ 11\ 10\ 00\ 11\ 00\ 01 & (\mathrm{x} \gg 1) \\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01 & \mathrm{0x5555} \\ \hline 01\ 01\ 01\ 00\ 00\ 01\ 00\ 01 & (\mathrm{x} \gg 1)\ \&\ \mathrm{0x5555} \end{array}$$
- $$\begin{array}{ll} 00\ 01\ 01\ 00\ 01\ 00\ 00\ 01 & \mathrm{x}\ \&\ \mathrm{0x5555} \\ 01\ 01\ 01\ 00\ 00\ 01\ 00\ 01 & (\mathrm{x} \gg 1)\ \&\ \mathrm{0x5555} \\ \hline 01\ 10\ 10\ 00\ 01\ 01\ 00\ 10 & (\mathrm{x}\ \&\ \mathrm{0x5555}) + (\mathrm{x} \gg 1)\ \&\ \mathrm{0x5555} \end{array}$$

Schnelle parallele Addition von Bits

10 11 11 00 01 10 00 11	x
01 01 01 01 01 01 01 01	0x5555
00 01 01 00 01 00 00 01	x & 0x5555

10 11 11 00 01 10 00 11	x
01 01 11 10 00 11 00 01	$(x \gg 1)$
01 01 01 01 01 01 01 01	0x5555
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555

10 11 11 00 01 10 00 11	x
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555
01 10 10 00 01 01 00 10	$(x - (x \gg 1)$ & 0x5555

```java
public static int bitCount(long i) {
  i = i - ((i >> 1)  & 0x5555555555555555L);
  i =    (i          & 0x3333333333333333L)
     + ((i >> 2)     & 0x3333333333333333L);
  i = (i + (i >> 4)) & 0x0f0f0f0f0f0f0f0fL;
  i = i + (i >> 8);
  i = i + (i >> 16);
  i = i + (i >> 32);
  return (int)i & 0x7f;
}

public static void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(Long.bitCount(t-1));
    bitset ^= t;
  }
}
```

Problem

- Keine bedingten Sprünge :-)
- Immer noch 17 Operationen / 1-Bit :-(
- Kann nur noch wenig optimiert werden

Problem

- Keine bedingten Sprünge :-)
- Immer noch 17 Operationen / 1-Bit :-(
- Kann nur noch wenig optimiert werden

Aber ...

- Es gibt eine völlig andere Lösung

Achtung Mathe!!!!111!!!!

Einige Definitionen

Kombinatorik

Kombinatorik ist das Teilgebiet der Mathematik, dass sich mit **endlichen** und abzählbar unendlichen Strukturen beschäftigt.

Alphabet

Eine Alphabet ist eine endliche Menge von Buchstaben z.B. $\{0, 1\}$ oder $\{a, b, c\}$.

Wort / Sequenz

Ein Wort oder Sequenz über einem Alphabet A ist eine Abfolge von Buchstaben aus A z.B. 10110101.

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

Beispiel

0011 ist ein De Bruijen Sequence $B(2, 2)$. Denn: $2^2 = 4$ und 0011, 0011, 0011, 0011 sind alle Sequenzen der Länge 2 über $\{0, 1\}$

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

Beispiel

0011 ist ein De Bruijen Sequence $B(2, 2)$. Denn: $2^2 = 4$ und 0011, 0011, 0011, 0011 sind alle Sequenzen der Länge 2 über $\{0, 1\}$

Beispiel

00010111 ist ein De Bruijen Sequence $B(2, 3)$. Denn: $2^3 = 8$ und 00010111, 00010111, 00010111, 00010111, 00010111, 00010111, 00010111, 00010111 sind alle Sequenzen der Länge 3 über $\{0, 1\}$

Na und?

Na und?

- Was wäre, wenn wir $B(2,6)$ finden könnten?

Na und?

- Was wäre, wenn wir $B(2, 6)$ finden könnten?
- $B(2, 6)$ wäre 64 Zeichen $\equiv$ 64 Bit lang
- $B(2, 6)$ enthält jede Teilsequenz der Länge 6 genau einmal
- Also jede Zahl zwischen 0 und 63 genau einmal

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten ...

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten …
- … haben wir eine eindeutige Zahl zwischen 0 und 63

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2, 6) \cdot x$ die obersten 6 Bits betrachten ...
- ... haben wir eine eindeutige Zahl zwischen 0 und 63
- Leider nicht in der richtigen Reihenfolge

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten ...
- ... haben wir eine eindeutige Zahl zwischen 0 und 63
- Leider nicht in der richtigen Reihenfolge
- Aber dafür kann man in eine Tabelle gucken

Beispiel für 8 Bit

$B(2,3) \cdot 2^i \equiv B(2,3) \ll i$

$B(2,3)$		x & -x			Markierte Bits
00010111	$\cdot$	00000001	=	00010111	0
00010111	$\cdot$	00000010	=	00101110	1
00010111	$\cdot$	00000100	=	01011100	2
00010111	$\cdot$	00001000	=	10111000	5
00010111	$\cdot$	00010000	=	01110000	3
00010111	$\cdot$	00100000	=	11100000	7
00010111	$\cdot$	01000000	=	11000000	6
00010111	$\cdot$	10000000	=	10000000	4

Lookup-Table

table:

0	1	2	3	4	5	6	7
0	1	2	4	7	3	6	5

Überraschung

Es gibt Beweise die zeigen: für jede Wahl von n und k gibt es DeBruijen-Sequenzen der Form $B(n, k)$.

Binär

0000001000011000101000111001001011001101001111010101110110111111

Hexadecimal

0x0218a392cd3d5dbfL

Hexadecimal

0x0218a392cd3d5dbfL
ist eine DeBruijen Sequenz der Form $B(2, 6)$

```
int table[64];
public static void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(table[(t * 0x0218a392cd3d5dbfL) >>> 58]);
    bitset ^= t;
  }
}
```

Muss nur einmal ausgerechnet werden

```
int table[64];
public static void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(table[(t * 0x0218a392cd3d5dbfL) >>> 58]);
    bitset ^= t;
  }
}
```

Problem

- Nur noch 7 Operationen / 1-Bit
- Aber eine davon potentiell seeeeehr teurer table lookup

Problem

- Nur noch 7 Operationen / 1-Bit
- Aber eine davon potentiell seeeeehr teurer table lookup

But ...

- ... we are still not done yet !!!!111oneeleven!!!!

Surprise: Hardwarhacking

Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal

Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung

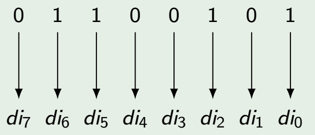
Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung
- Damit Reihenfolge der Bits doch wie aus DeBruijen-Sequenz

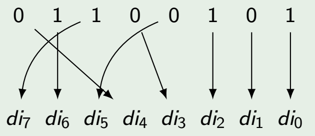
Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung
- Damit Reihenfolge der Bits doch wie aus DeBruijen-Sequenz
- Damit kein Table-Lookup mehr nötig

Beispiel (8 Bit)



Beispiel (8 Bit)



```java
public static void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process((t * 0x0218a392cd3d5dbfL) >>> 58);
    bitset ^= t;
  }
}
```

Strike

Strike

- 6 Operationen / 1-Bit

Strike

- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction

Strike

- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction
- Falls alle Bits gesetzt sind $6 \cdot 64 = 384$ Operationen

Strike

- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction
- Falls alle Bits gesetzt sind $6 \cdot 64 = 384$ Operationen
- Damit:

  Worst Case Performance dieses Codes

  $\equiv$ Best case Performance des Codes mit dem wir angefangen haben

Glaub ich nicht!!!111einself!!

Gimme the numbers!!!111einself!!